Electronic Communications of the EASST
Volume 19 (2009)

Proceedings of the Second International DisCoTec Workshop on Context-Aware Adaptation Mechanisms for Pervasive and Ubiquitous Services (CAMPUS 2009)

An Adaptation Reasoning Approach for Large Scale Component-based Applications

Mohammad U. Khan, Roland Reichle, Michael Wagner, Kurt Geihs, Ulrich Scholz, Constantinos Kakousis, and George A. Papadopoulos

12 Pages

Guest Editors: Romain Rouvoy, Michael Wagner
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

An Adaptation Reasoning Approach for Large Scale Component-based Applications

Mohammad U. Khan[1], Roland Reichle[1], Michael Wagner[1], Kurt Geihs[1], Ulrich Scholz[2], Constantinos Kakousis[3], and George A. Papadopoulos[3]

[1]University of Kassel, Germany, {khan, reichle, wagner, geihs}@vs.uni-kassel.de
[2]European Media Laboratory GmbH, Germany, ulrich.scholz@eml-d.villa-bosch.de
[3]University of Cyprus, Cyprus, {kakousis, george}@cs.ucy.ac.cy

Abstract: There is a growing demand for context-aware applications that can dynamically adapt to their run-time environment. An application offers a collection of functionalities that can be realized through a composition of software components and/or services that are made available at runtime. With the availability of alternative variants of such components and/or services that provide the basic functionalities, while differ in extra-functional characteristics, characterized by quality of services (QoS), an unforeseen number of application variants can be created. The variant that best fits the current context is selected through adaptation reasoning, which can suffer from the processing capabilities of resource-scarce mobile devices, especially when a huge number of application variants needs to be reason about. In this paper, we present a reasoning approach, which provides a meaningful adaptation decision for adaptive applications having a large number of variants within a reasonable time frame. The approach is validated through two arbitrary applications with large number of variants.

Keywords: self-adaptation, ubiquitous computing, adaptation reasoning, variability, scalability, utility function

1 Introduction

Mobile and ubiquitous computing introduce a growing demand for applications that are able to adapt to dynamically changing execution environments, resources and user preferences. For example, applications may want to react dynamically to fluctuations in network connectivity, battery capacity, appearance of new devices and services, and to a change of user profiles and their choices. Such adaptations are handled by automatically choosing a different variant of the application that provides the same basic functionality with a changed quality of service. [1]

For component-based applications, with the option of integrating external services, application variants can be created according to a variability model; similar to what is practised by the product-line community [1]. An application is composed of components and/or services, where each component/service can have a number of different variants[1]. Therefore, the total number of application variants is a product of the variants of each of its constituent components. In a ubiquitous computing environment, components and services can appear and disappear at runtime. This fact advocates against a static architecture of the application, which requires that at least one variant of a constituent component must be present. Therefore, the architecture of the application can also evolve, with the possibility of using a completely different set (and number) of components and services to realize the application. For such an

[1] The details on constructing the variability model are presented in section 2.

evolving architecture, possible application variants can not be foreseen at design-time. The variability model of creating application variants suffer from the possibility of combinatorial explosion, because in worst case the number of application variants is a product of number of variants of individual components.

Runtime adaptation involves detecting and keeping track of available components, services and their meta-information, selecting a variant through adaptation reasoning based on the current context and then (re)configuring the composition to realize the selected variant of the application. Adaptation reasoning performs the selection of the application variant that fits best to the current context and resource situation. This step suffers the most from the combinatorial explosion, because ideally the fitness of each variant needs to be checked. In the utility function-based approach [3], the fitness is calculated by evaluating utility functions. In our previous work [4], we have presented the concept and methodology for supporting such adaptations. However, test results showed that the adaptation reasoning technique presented at [4] works well only for applications with a limited number of variants; but becomes practically useless for applications, which may offer a huge number of variants. Such a reasoning technique requires evaluating not only the utility separately for each application variant; but also property predictors [5] that compute its quality of service requirements and used by the utility evaluation. Especially for resource scare mobile devices, such computational effort makes meaningful adaptation infeasible.

In this paper, we present a reasoning approach which is stable against combinatorial explosions and thus can provide adaptation reasoning within a meaningful time-frame, even for devices with low computational resources. The rest of the paper is organized as follows: Section 2 describes a technique of creating application variants with the help of a simple example. Section 3 presents the reasoning approach and section 4 provides some evaluation results for arbitrarily large application architectures. Section 5 compares the work with the state of the art and in section 6, the paper is concluded summarizing its achievements and pointing to future works.

2 Application Variability Model

The application architecture is created based on a variability model and thus it offers the possibility of creating different variants of the application that differ in extra-functional characteristics. When there is a significant context change, the middleware evaluates and compares all available application variants based on different QoS-metadata associated to the involved component realizations. Thus we consider applications that are developed with a QoS-oriented component model, which defines all reasoning dimensions used by the planning-based middleware to select and deploy the component implementation providing the best utility. The utility of a component utilization is computed using a developer-defined utility function. Such utility functions evaluate the fitness of a particular component variant based on the QoS-properties required by its realization and that provided by the current context [6].

Application Conceptual Metamodel

An Application Type is viewed as a Component Type that can have different realizations (Figure 1), where an application is such a realization of the application type. The meta-information of a certain realization is described using Plans. A component type can be realized by a single component, or by a composition of components, resulting in the concepts of atomic

and composite component types, respectively. Corresponding to the atomic and composite component types, there are two types of Plans: Atomic Realization and Composite Realization. In addition to the QoS-properties, an Atomic Realization Plan describes an atomic component and contains a reference to the class or the data structure that realizes the component. The Composite Realization Plan describes the internal structure of a composite component by specifying the involved Component Types and the connections between them.

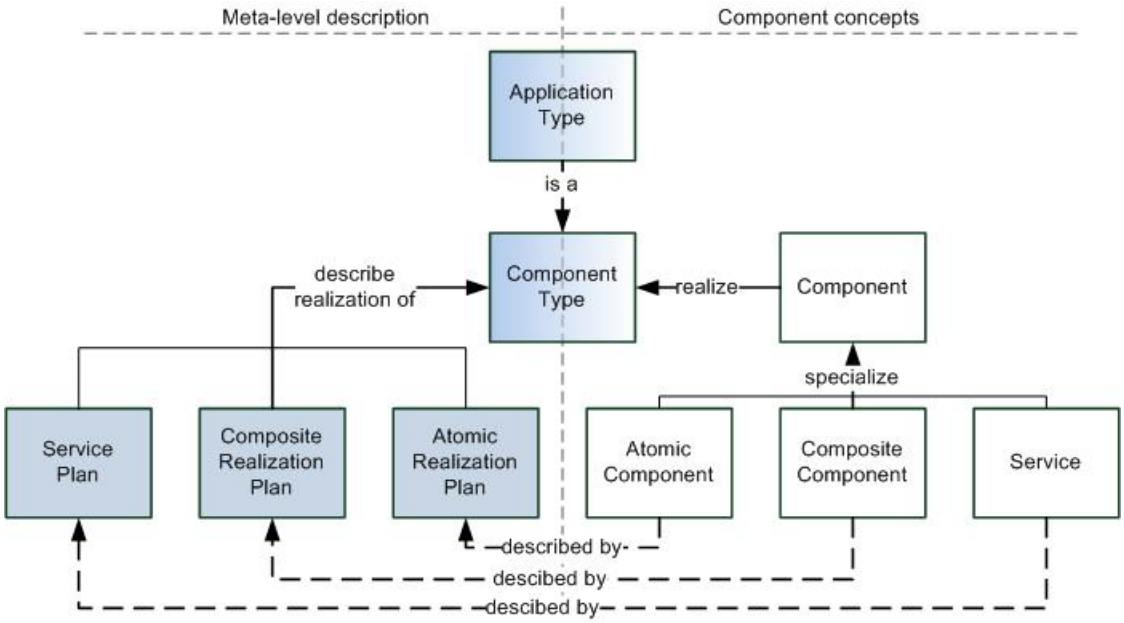

Figure 1: Application concepts and meta-information

Variation is obtained by describing a set of possible realizations of a Component Type using Plans. In order to create a possible variant, one of the Plans of a Component Type is selected. If the Plan is a Composite Realization Plan, it describes a collaboration structure of further Component Types, which in turn are described by Plans. Now we proceed by recursively selecting one realizing Plan for every involved Component Type. The recursion stops if an Atomic Realization Plan is chosen. Therefore, by resolving the variation points we create application variants that correspond to a certain composition of components depending on the plans that are chosen for each of the Component Types.

With service-based adaptation a part-functionality may be provided through a dynamically discoverable and accessible service. Thus, compositional adaptation is extended by taking a service as a possible realization of a Component Type. To do so, the QoS-properties, interfaces, and binding information have to be included in a corresponding plan. Service plans are created at runtime, based on the discovery of a service and along with Composition and Atomic Plans, they are also considered during the adaptation reasoning.

Runtime Creation of Application Variants

Application types, components, component types and plans are combined together in OSGi bundles that are deployed on an adaptation middleware [6]. A distributed environment can include any number of nodes in the middleware domain that are reachable or not due to

changes in the network. Bundles can be deployed on these nodes at any time, such that components, component types, and plans can appear and disappear unpredictably. When a new bundle is deployed on an existing node or on a node entering the domain, the middleware collects the information about the deployed application types, component types, plans and components. References to these component types and plans are stored in respective repositories. The middleware provides the repository service and therefore such repositories can as well be distributed over several nodes. The relation between a component type and a plan is established through their information model. When a node leaves the domain, the bundles deployed on them are removed from the repository, eventually removing the bundle contents like plans, component types etc. Thus the middleware keeps an up-to-date trace of all the available component types and plans.

Another task of the middleware is to discover services that realize different component types, marked as realizable through services. Each discovered service has a service description based on which a service plan is created and registered in the plan repository. Thus, at runtime the application architecture corresponds to a variability hierarchy containing component types and their realization plans.

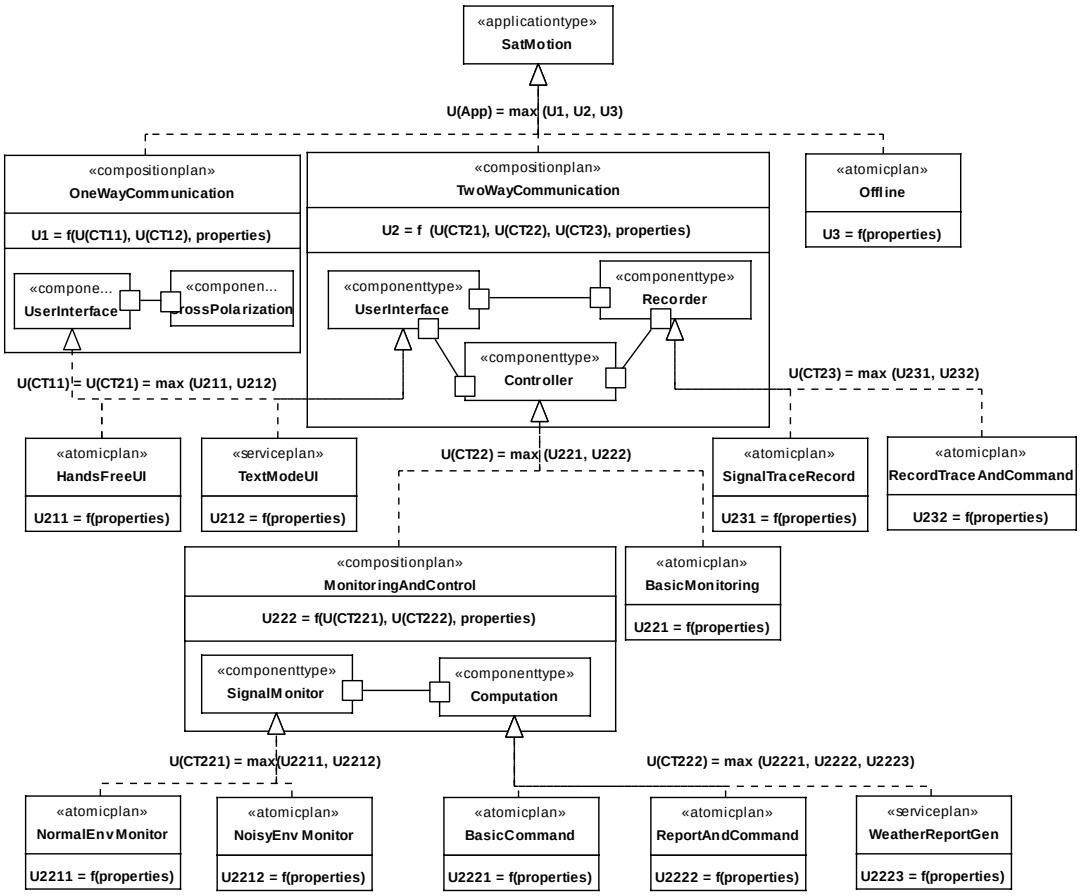

Figure 2 A variability hierarchy consisting of component types and plans

We illustrate the approach with a simplified version of a proof-of-concept application called SatMotion [7]. The purpose of this application is to aid an installer of satellite antennas to align them depending on the network characteristics of the execution environment. The SatMotion application can run in a number of different modes. For example, in the Two Way mode, besides receiving the trace of the antenna signal, it can communicate with a server and send command signals requesting different antenna parameters. In the One Way mode, it can only receive the signal trace but cannot send commands to the server. In the Offline mode, it can only playback and analyze the recorded signal. The details of the application are beyond the scope of the paper and are not required to understand the variability architecture.

Figure 2 presents an example of a variability hierarchy for the SatMotion application created at runtime, consisting of component types and their realizing plans. These component types and plans may be deployed separately on different nodes as part of different bundles. At runtime, let us consider that the SatMotion application type has three realization plans; two of them are composition plans and the other one, the Offline realization, is an atomic plan. The OneWayCommunication composition plan realizes a variant that has a composition of two component types UserInterface and CrossPolarization. The first one has two realizing plans; but no plan is available at runtime for the CrossPolarization component type. The plan TwoWayCommunication has a composition specification consisting of three component types, UserInterface, Recorder and Controller. The component types UserInterface and Recorder have only atomic and service plans, while the Controller component type has one atomic plan BasicMonitoring and one composition plan MonitoringAndControl. Each of the component types SignalMonitor and Computation has atomic and/or service realization plans.

In such a variability hierarchy, an atomic plan or a service plan indicates the bottom level of the variability tree. So, they can be always applied to realize the component type. However, a composition plan is only applicable if each of the component types involved in its composition specification has at least one applicable plan. For example, the OneWayCommunication plan is not applicable because there is no plan for the realization of the CrossPolarization component type used in its composition. Therefore, the application can be realized either using the Offline atomic plan or the TwoWayCommunication plan. Being a composition plan, the latter contains a composition of components. The UserInterface and the Recorder component types have two atomic plans each to choose from, while the Controller component type has one atomic plan and one composition plan. Such options of alternating plans offer variability in composing an application.

3 Runtime Reasoning

We have developed a middleware to provide the runtime support of adapting the application [6]. The number of application variants (section 0) increases rapidly with the number of component types participating in a composition. Though this increase is not prominent for a very simple architecture like that presented in Figure 2, it becomes an issue of great concern quite rapidly. For example, a composition plan having 6 different component types, where each of the types has 10 different atomic plans, will have one million ($1M = 10^6$) variants for this particular composition plan only. Selecting the best-fit variant calculating the utility of each of such variants, resulting from a combinatorial explosion, is a time consuming task and often fails to provide a solution within a reasonable time frame. Therefore, we have developed a new reasoning approach, looking at the problem from a different perspective to make it free

of such combinatorial explosions. We first present the reasoning approach and afterwards, we explain the integration of related aspects like checking resource limits along with reacting on context changes.

The Reasoning Approach

A component or a service has a certain utility for a particular context based on its QoS-properties. The utility can be evaluated at runtime by a developer defined utility function. An application is composed of components and services. Moreover, other properties, like the communication among different components may influence the fitness of a particular component composition. Therefore, it is assumed that the utility of the application can depend on the utility of its constituent components as well as such properties. We also assume that a higher utility of a constituent component will contribute to a higher utility for the overall application.

In the application variability model (section 2), each atomic realization plan and service plan has a set of QoS-property specifications that indicates the quality of service characteristics required from the context and resources for the component or service to be usable. A utility function takes those requirements into account and computes a utility for the realizing plan by comparing them with the context and resource characteristics of the run-time environment.

In addition to a utility function, a composite realization plan contains a composition of component types. Let us consider that $\mathbf{CT} = \{CT_1, CT_2, \ldots, CT_n\}$ is the set of component types that is involved in a composition **C**. For all $CT_i \in \mathbf{CT}$, there exist sets $\mathbf{A} = \{a_1, a_2..., a_p\}$, $\mathbf{B} = \{b_1, b_2..., b_q\}$, …, $N = \{n_1, n_2..., n_z\}$ where, $a_i \in$ Realization Plans of CT_1, $b_i \in$ Realization Plans of CT_2, …, $n_i \in$ Realization Plans of CT_n.

Let us denote the utility of the realization plan a_i as Ua_i. The utility of each chosen realization plan for a component type contributes to the overall utility of a particular composition, and eventually the composite realization plan, of which the component type is a part of. If $U_{CT(ai)}$ denotes the contribution to utility for the composition when the realization a_i is chosen, then it is assumed that

$$\text{(I) } Ua_i \geq Ua_j \Rightarrow U_{CT(ai)} \geq U_{CT(aj)};\ \forall\ a_i, a_j \in \mathbf{A}$$

The maximum utility available for the realization of a particular component type U_{CT1} is,

$$\text{(II) } U_{CT1} = \max\ (U_{CT(a1)}, U_{CT(a2)}, \ldots, U_{CT(ap)})$$

In order to derive the maximum utility of the composition, U_c, a function satisfying (I) can be defined as

$$\text{(III) } U_c = f(U_{CT1,} U_{CT2, \ldots.,} U_{CTn,} U_{prop})$$

where, U_{prop} is the contribution of properties (non-related to the individual components, rather related to the composition, communication among components etc.) to the utility.

In general, the equation (III) can take any form, given that for each realization plan a_i , equation (I) is also maintained. A special case of equation (III) can be represented as follows:

$$\text{(IV) } U_c = \sum_{i=1}^{n} w_i U_{CTi} + w_{n+1} U_{prop}$$

where, $\sum_{i=1}^{n} w_i = 1.0$ and each w_i indicates the relative importance (weight term) of a component type within a composition, as assigned by the developer while specifying the realization plan.

In order to illustrate the approach with the help of the example architecture presented in Figure 2, let us consider that the QoS-properties of the NoisyEnvMonitor plan are as follows:

```
Memory = 100;
EnvNoise = HIGH;
NetworkType = WiFi;
```

Based on these QoS-properties, a utility function can be defined follows:

```
U2212 = 0; if EnvNoise = LOW

        (
          (
             1.0; if context.Memory ≥ 100
             1.0 – (100 – context.Memory)/100; otherwise;
          )
        + (
             1.0; if context.NetworkType = WiFi
             0.0; if context.NetworkType = None
             0.5; otherwise
          )
        )/2.0; otherwise
```

A runtime value of Memory = 90 and NetworkType = WiFi, with a HIGH EnvNoise will result in a utility value of

```
U2212 = ((1.0 – (100-90)/90) + 1.0)/2.0 = 0.95
```

U2211 can be calculated similarly according to another function and if U2212>U2211, the NoisyEnvMonitor plan is chosen for the realization of the SignalMonitor component type. Then U(CT221) = max(U2211, U2212) = 0.95

The utility for the Computation component type, U(CT222) can be computed in the same manner. For this example, let us consider that U(CT222) = 0.8

Now, in the simplest case, let us assume that the utility of the MonitoringAndControl plan has a contribution of 50% from U(CT221), 30% from U(CT222) and 20% from its properties. The property contribution can be expressed the same way using a function. Let us presume that the value is 0.7. Then,

```
U222  = 0.5xU(CT221)+0.3xU(CT222)+0.2x0.7
      = 0.5x0.95+0.3x0.8+0.2x0.7 = 0.855
```

Following this approach, the utilities U2 for the TwoWayCommunication and U3 for the Offline plan can be calculated and the one providing the highest utility is selected to run. While realizing the application, the chosen plans at different levels are considered to instantiate the components and/or to bind to the services.

It is to be noted that the successful application of the approach depends on a few reasonable assumptions (e.g., equation (I)); but it does not apply to utility functions that violate these

assumptions. For example, when the utility of a constituent component reduces the overall utility of the composition (though, unlikely) or when the utilities of a component influences the utility of another component in the composition, then the assumption becomes invalid.

Fitting within the Resource Constraints

In our approach, each running application is allowed to use a certain amount of resources, assigned to it by an underlying middleware or operating system. Therefore, the application variant chosen (by applying the reasoning approach of section 0) might not be practically realizable. This problem demands a check of resource constraints of the chosen variant against the runtime availability of the required resources. If such constraints are not met, another variant must be chosen that will probably provide lower utility; but fits within the resource constraints.

Ideally, resource constraints could be checked for each of the variants before checking for their utilities; but that process would suffer from the combinatorial explosion, which we would like to avoid. Therefore, we first find a variant by applying the reasoning approach and then apply a local search mechanism to find a variant that provides a feasible solution satisfying constraints for each of the resources with the minimum sacrifice to the utility.

The search is performed once for each of the resources. The target is to use a different variant for the individual components until the resource constraints are met. The first step in the search mechanism is to select the starting point among the chosen components for the application composition. The component that requires the most resource can be a reasonable target, because a second variant of that component would most probably release an appreciable amount of resources, in a way to speed up the search. A second choice would be to start with the least important component so that replacing it with its second best variant would not result in much loss of utility. Both of these choices have their pros and cons and a combination of them would suggest using the ratio resource needs to importance as the guiding factor to select the starting point.

For the starting component, an alternative is chosen, which consumes less resource than the previously chosen one, while provides the highest utility among the remaining options. For example, in Figure 2, if ReportAndCommand plan (for corresponding component) was initially chosen; but fails in resource constraint, then the one between BasicCommand and WeatherReportGen provides the higher utility is chosen in this step, given neither of them requires more resource than the ReportAndCommand plan. If the resource saved because of selecting this new variant is still not sufficient to meet the resource constraint, then we proceed with the next component. For example, we could now replace NoisyEnvMonitor by NormalEnvMonitor, if the later requires less resource, though provides a slightly lower utility. The search mechanism also takes into account the cases, where a composite realization plan may appear as an alternative to an atomic realization plan. For example, if the BasicMonitoring atomic realization plan were initially chosen; but fails in resource constraint, then the search will consider the MonitoringAndControl composite realization plan to find the best configuration applying the reasoning approach of section 0, given this configuration collectively requires less resource than the component corresponding to the BasicMonitoring plan.

The approach has the limitation that in extreme cases, we might have to sacrifice utilities in

great extent; but it helps avoiding the combinatorial explosion and therefore fits well within the reasoning approach. Therefore, it will provide a feasible (satisfying architectural and resource constraints) solution, if any, within a time frame acceptable to the user of the application

Integrating the Effect of Context Changes

In order to further improve the adaptation reasoning performance, we selectively reason about composition plans incorporating component types that have been affected by the specific context change which has triggered the current adaptation process. Through realization plans each component type explicitly defines its context and resource dependencies. Thus, whenever a context change triggers an adaptation, the adaptation reasoner can omit recalculating the utility values for the component types not being associated with the changed context elements. Of course, such an optimization technique requires keeping track of the lastly calculated utility score, for each component type or composition plan. This technique will enable evaluation of composition plan utilities, without recalculating utilities of unaffected component types.

To illustrate the effect of context-based adaptation reasoning, we provide an example based on the variability hierarchy model depicted in Figure 2. For the SatMotion application it is reasonable to assume that only the SignalMonitor component type is associated with the EnvNoise context element. Thus, a possible context change to the ambient noise level may only affect the utilities of SignalMonitor and of its realizing components: NormalEnvMonitor and NoisyEnvMonitor. Therefore, if we remember the previous utility score for the Computation component type, we can evaluate U222 without recalculating U2221, U2222 and U2223. Based on the same reasoning, we may skip U221, U231, U232, U211, U212 and U3 (U1 is omitted anyway since OneWayCommunication is not applicable) which leads us to a total gain of 69.2 percent (i.e., only 4 out of 13 applicable utility functions were recalculated).

Advantages of the Approach

In this approach, the number of times the utility function has to be evaluated corresponds to the number of 'applicable' plans. In the simple example of Figure 2 we need to evaluate it only for 13 times (the OneWayCommunication plan is not applicable), while in the case of the one million variant as mentioned at the start of section 3, we have to evaluate the utility for only 60 times. Thus, the approach becomes stable against scalability and more advantageous as the number of variants increases.

Moreover, our experiences show that the specification of proper utility functions is a big challenge requiring lot of intuition for the developer, especially for the one covering the complete application. However, this task is simplified, if they have to specify utility for individual components.

4 Evaluation Results

The superiority of the proposed evaluation approach is established by the fact that it requires the evaluation of the utility function once for each plan. Therefore, with the increase of number of plans in the variability hierarchy, the evaluation time should also increase linearly, unlike the number of possible application variants, which may increase exponentially due to combinatorial explosions.

Figure 3 Reasoning time on Windows Mobile Emulator

The effect is negligible for simple cases like what we have presented in Figure 2. Therefore, we have tested the scalability effect with two large scale examples; the first one comprises ~2M application variants while a second set up consists of a total of around 15M application variants. Please note that these examples are no real life applications and therefore, the used components only helps printing some messages to denote the selected variant. The focus is on the speed of the adaptation reasoning, rather than on the application functionalities.

The evaluation results are presented in Table 1. The reasoning time on a Desktop PC is negligible, while for a Mobile device[2], it is only a few seconds. More importantly, the reasoning time does not increase drastically, even though there is a huge increase in the possible number of application variants. As a comparison to such numbers, a Brute-force reasoning approach, which calculates utility separately for each application variants, takes almost 14 minutes on a Desktop PC for the 2M (million) variant example.

Table 1: Evaluation result

Device	**Reasoning Time**	
	for 2M variants	*for 15M variants*
Desktop PC WinXP, 3GHz, 1GB RAM	<20 ms	< 20 ms
ARM 920T Device Emulator Windows Mobile 5, 62MB	~ 1 s	1.5 – 2.0 s

These results show that this new approach can be applied to reason about the adaptation within an acceptable time, even for large scale applications having a huge number of application variants. This improvement is particularly important where the application architecture can dynamically evolve at runtime, because the availability of new components, component types, realization plans, and services may result in growing the number of possible variants quite rapidly.

5 Related Work

Development methodologies, platforms and middleware supporting dynamic adaptation of context-aware applications on mobile computing devices have been studied extensively during the last decade. Some early approaches [11] provided support for adaptations foreseen during the design of the application. They usually have only a limited number of adaptation options and therefore, they are easier to handle and usually a policy-based approach [12] for the adaptation decision is sufficient.

[2] We have used Windows Mobile 5 Emulator on a Laptop to support using PhoneME with Knopferfish, which we could not run on a real device having WM 2003. The performance of an emulator depends also on the host device (Laptop) and from our experience with WM 2003, Emulators usually perform worse than a real device.

The complexity is increased when the need for runtime adaptation arises. The policy-based approach becomes inapplicable, because the context and resource situation can not be fully predicted. The utility function-based approach of adaptation decision might be applied in such cases [3][4][13]. As for example, SAFRAN [9] is a framework for building self-adaptive, component-based applications that separates the application logic from the adaptation. It is very high level and, in principle, allows for the implementation of techniques similar to distributed utility-based adaptation by allowing each component to decide upon which reconfiguration to operate. In [3], utility functions express service level attributes to dynamically allocate resources in an autonomic data centre system. Works like [4] and [13] are predecessors of our work, applied in the same application areas, though they did not provide solutions for taking care of scalabilities.

Utility-function based adaptation policies get rid of the shortcomings of predicting all adaptation decisions at design-time; however, they introduce the need for evaluating utilities at runtime. One of the shortcomings of the utility-function based solutions of adaptation decision is the scalability [8], especially when they are combined with the variability approach of creating the application architecture. The computation effort may increase exponentially with the number of variation points. Some works [8][10] have tried to apply some heuristics in simplifying that computation. However, the computation effort is still not linear with the variation points and the need for property predictors has added to extra computation. The approach presented in this paper can greatly aid the solution to such problems.

6 Conclusions

Adaptations of applications running in a ubiquitous computing environment require an application architecture created through the composition of components and services available at runtime. Such an adaptation process involves a number of tasks like retrieving information about components and services, building the application architecture at runtime, reasoning about them and configuring the best-fit composition. In this paper, we have presented an approach that aids adaptation reasoning, which would otherwise suffer from the scalability problem due to the combinatorial explosion of the number of application variants.

The presented approach provides a solution for adaptation reasoning, which is stable against the scalability and can be applicable, even when the number of application variants becomes quite huge. The combinatorial explosion is avoided by considering the utilities of each realization plan separately than combining them together to find the complete application variant before reasoning. The superiority of the approach is supported theoretically as well as through practical test cases. It also aids the application developers by easing the process of defining utility functions and QoS-properties for their components.

The solution also integrates support for checking resource constraints, which have to be dealt with during the reasoning of adaptation. Besides, we are currently updating our solution to architectural constraints [7], which would limit choosing among different realization plans by specifying constraints, to fit within this new adaptation reasoning approach.

However, the effectiveness of the approach depends on a few assumptions. These assumptions may not be applicable in cases, where the utility of a composition does not positively depend on the utilities of its constituent components. In the future, we are going to investigate more on such cases and improve the solution as necessary.

Figure 4 Reasoning time on Windows Mobile Emulator

7 References

[1] Sinnema, M., Deelstra, S., Nijhuis, J., Bosch, J., "*COVAMOF: A Framework for Modelling Variability in Software Product Families*," Proc. Third Software Product Line Conference, Springer, 2004, pp. 197–213.

[2] Floch, J., Hallsteinsen, S., Stav, E., Eliassen, F., Lund D., and Gjørven, E. *Using architecture models for runtime adaptability*. IEEE Software 23(2):62--70, 2006.

[3] Walsh, W.E., et al. *Utility Functions in Autonomic Systems*. In proceedings of First International Conference on Autonomic Computing (ICAC'04), pp. 70-77. 2004.

[4] Geihs, K., Barone, P., Eliassen, F., Floch, J., Fricke, R., Gjørven, E., Hallsteinsen, S., Horn, G., Khan, M. U., Mamelli, A., Papadopoulos, G.A., Paspallis, N. Reichle, R., Stav, E., *A Comprehensive Solution for Application-Level Adaptation*. Journal on Software Practice and Experience, 2009; No. 39; pp. 385 - 422.

[5] Brataas, G., Floch, J., Rouvoy, R., Bratskas, P., and Papadopoulos, G.A. *A Basis for Performance Property Prediction of Ubiquitous Self-Adapting Systems*. In: Proceedings of the International Workshop on the Engineering of Software Services for Pervasive Environments (ESSPE'07), pp. 59–63, Dubrovnik, Croatia, ACM. AICPS. 2007.

[6] Rouvoy, R., Barone, P., Ding, Y., Eliassen, F., Hallsteinsen, S., Lorenzo, J., Mamelli, A., and Scholz, U. *MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments*. Chapter in book on Software Engineering for Self-Adaptive Systems (SEfSAS), LNCS, Springer, 19 pages, 2009. Accepted for publication.

[7] Khan, M.U., Reichle, R., and Geihs, K. *Architectural Constraints in the Model-Driven Development of Self-Adaptive Applications*. IEEE Distributed Systems Online, vol. 9, no. 7, 2008.

[8] Scholz, U. and Rouvoy, R. *Divide and Conquer - Organizing Component-based Adaptation in Distributed Environments*. Communications of the EASST, 11, 2008.

[9] David, P.-C. and Ledoux, T. *An Aspect-Oriented Approach for Developing Self-Adaptive Fractal Components*. In 5th International Symposium on Software Composition. LNCS 3089, pp. 82–97. Springer, 2006.

[10] Alia, M., Horn, G., Eliassen, F., Kahn, M.U., Fricke, R., and Reichle, R. *A Component-based Planning Framework for Adaptive Systems*. The 8th International Symposium on Distributed Objects and Applications (DOA), Oct 30 – Nov 1, 2006. Montpellier, France.

[11] Mätzel, K. and Schnorf, P. *Dynamic Component Adaptation*. In ECOOP 2002 Workshop Reader, LNCS 2548. Springer, 2002.

[12] Lutfiyya, H., et. al. *Issues in Managing Soft QoS Requirements in Distributed Systems Using a Policy-Based Framework*. In proceedings of 2nd International Workshop on Policies for Distributed Systems and Networks (POLICY'01). Springer, p. 185-201. 2001.

[13] Rouvoy, R., Eliassen, F., Floch, J., Hallsteinsen, S., and Stav, E. *Composing Components and Services using a Planning-based Adaptation Middleware*. In proceedings of the 7th International Symposium on Software Composition (SC'08), LNCS 4954. Springer, 2008.